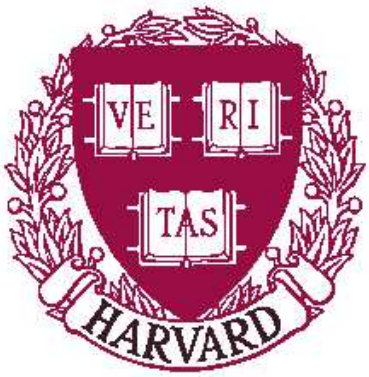
Fast visualization of depth contours using graphics hardware

Fast Visualization of Depth Contours Using Graphics Hardware

Ian Fischer
and
Craig Gotsman

TR-23-05



Computer Science Group
Harvard University
Cambridge, Massachusetts

Fast Visualization of Depth Contours Using Graphics Hardware

Ian Fischer
Harvard University

Craig Gotsman
Technion – Israel Institute of Technology

Abstract

Depth contours are a well-known technique for visualizing the distribution of multidimensional point data sets. We present an image-space algorithm for drawing the depth contours of a set of planar points. The algorithm is an improvement on existing algorithms based on the duality principle from computational geometry, implemented with 3D graphics rendering techniques. Our improvement takes advantage of properties of the dual arrangement of the input point set to significantly reduce the amount of computation, thus is asymptotically faster than its predecessors.

Categories and Subject Descriptors: G.3 [Probability and Statistics]: Statistical software; I.3.5 [Computation Geometry and Object Modeling]: Geometric algorithms, languages, and systems.

1. Introduction

Depth contours are a well-known technique for visualizing the distribution of multidimensional point data sets [MRR*2001]. Given a set P of n points in R^d, the *location depth* of a point $q \in R^d$ relative to P describes, intuitively, the relationship of q to the distribution of the points in P. The associated concept of *depth contour* is the locus of all points with the same fixed location depth. Drawing these nested depth contours helps to express the distribution of the points in a visual manner. In particular, the *Tukey median* is the centroid of the deepest such contour. See examples in Figure 1. Location depth and depth contours have applications in robust statistics, hypothesis testing, and some areas of cell biology. In this paper we will deal with the planar case, i.e. d=2. We start with some formal definitions.

Definition: Let $P = \{p_1, p_2, \ldots, p_n\}$ be a finite set of points in R^2 and q be an arbitrary point in R^2. The *location depth* of q relative to P, denoted by $ld_P(q)$, is the minimum number of points of P lying in any closed half plane determined by a line through q. ◊

$ld_P(q)$ is an integer in the range $\{0,..,n\}$. It is 0 when q lies outside the convex hull of P and n when all points coincide with q.

Definition: Let $P = \{p_1, p_2, \ldots, p_n\}$ be a finite set of points in R^2. The *k-th depth contour* of P, denoted by $dc_P(k)$, is the boundary of the set of all points q in R^2 with $ld_P(q) \geq k$. ◊

The k-depth contour is sometimes called the *k-hull*, and points on $dc_P(k)$ can be shown to be *k-splitters* of P, because every line through q has at least k points of P lying on or above it and at least k points of P lying on or below it.

See Figure 1 for an example of the depth contours of a point set. It is easy to verify a number of simple properties of depth contours:

1. $dc_P(k)$ is a convex polygon for any k.
2. $dc_P(1)$ is the boundary of the convex hull of P.
3. $dc_P(k+1) \subseteq dc_P(k)$.

It is less obvious to show that there will always be a depth contour of depth $\lfloor n/3 \rfloor$ but not necessarily one of depth $\lfloor n/2 \rfloor$. Note that depth contours are different from the so-called polygonal *onion layers* of a point set because vertices of onion layers are always points of P, while vertices of depth contours are not necessarily so.

2. Computing Depth Contours

2.1 The geometric algorithm

Miller *et al* [MRR*2001], improving results of Cole *et al* [CSY87], present an algorithm for computing the depth contours and related entities for a set of points. The algorithm itself is fairly simple, and makes extensive use of duality [MRR*2001], as is common in many computational geometric algorithms. Duality associates points and lines in the primal plane with corresponding lines and points in the dual plane. The algorithm proceeds as follows: Given a set P of points, map them to their dual arrangement of lines. Then apply topological sweep to find the planar graph of the arrangement and label the vertices of the arrangement with their *levels* – the number of dual lines above them. The *depth* of a vertex is $\min(level(v), n\text{-}level(v)+1)$. Finally, for a given k, compute $dc_P(k)$ by finding the lower and upper convex hull of the vertices at depth k. Each such vertex corresponds to a half-plane in the primal plane, and $dc_P(k)$ is the boundary of the intersection of these half-planes (which might be empty, in which case $dc_P(k)$ does not exist).

Note that a k-splitter is a primal point whose dual line separates the dual arrangement vertices having level k and those having level n-k+1. Thus a single k-splitter may be computed by solving a linear program with a number of constraints equal to the number of dual vertices, which could be $O(n^2)$.

The complexity of this geometric algorithm is $O(n^2)$ time and $O(n^2)$ space, which has been shown to be optimal. A related algorithm for computing the location depth of a single query point in $O(\log^2 n)$ time follows, and this is also optimal. The so-called *bag plot* of the points, which is the convex region containing no more than half the points, may also be computed in $O(n)$ time once the depth contours have been computed. Finally, the Tukey median can be easily computed once the deepest depth contour is known. See Figure 1.

2.2 The image-space algorithm

For large n, an $O(n^2)$ time geometric algorithm could be prohibitive, especially in an interactive application where the point set is dynamic. To address this, Krishnan *et al* [KMV2002] present an image space algorithm to approximate all the depth contours of a given point set. Essentially, it takes advantage of the graphics hardware to *draw* an image of the depth contours as a set of colored pixels. Thanks to this, their algorithm outperforms the geometric algorithm by at least one order of magnitude. But this efficiency comes at the price of losing sub-pixel information. However, as we shall see later, this error is minimal and the results are still quite acceptable.

The image-space algorithm is based on the same duality principle as the geometric algorithm. It consists of two phases. In the first phase, the input point set P is scan-converted to lines in the dual image plane. Since the dual plane is discrete, it is possible to compute the *level* of each pixel. This information is used in the second phase to create the depth contours. As opposed to the geometric algorithm, the image-space algorithm saves time by not explicitly computing convex hulls.

In more detail, the algorithm proceeds as follows. Without loss of generality, assume that each point in P is located inside the square of edge length two centered at the origin. In the first phase, transform each point of P to its line in the dual plane, and define the *level* of a point q in the dual plane to be the number of lines below or passing through it, and the *depth* of q to be $min\{n–\text{level}(q),\text{level}(q)\}$. Because our dual plane is of finite size, it is necessary to guarantee that all intersection points of the lines lie in this finite region. Krishnan *et al* claim that by using two separate bounded dual planes of size 2×4 – bounded dual 1 (BD1) maps (p_x, p_y) to $y = -p_x x + p_y$ and bounded dual 2 (BD2) maps (p_x, p_y) to $y = -p_y x + p_x$ – all intersections must lie in one of these two dual planes. Thus, in practice, the algorithm runs on both bounded duals. The level of each point in the dual plane is computed by drawing each dual line in turn and incrementing the region above the dual line by one in the stencil buffer. This is accomplished by drawing the entire half-plane lying above the dual line, which causes all the pixels on or above the line to be incremented by one if the stencil operation is set to increment. The dual lines themselves are drawn into a separate buffer in order to keep track of which pixels actually contain lines. See Figure 2 for an example of this.

Phase two uses the knowledge of the level of each point in the dual plane to compute the depth contours. The image of the dual lines is scanned, and for each point q on a dual line, they render the corresponding primal line at fixed z-depth $min\{n–\text{level}(q), \text{level}(q)\}$ as a colored 3D graphics primitive using the z-buffer. Since n is fixed and the value level(q) is available in the stencil buffer, the appropriate depth is easily determined. The hardware depth test is set to LESS while rendering these primal lines, and the line color should be distinct for each depth. The resulting rendered image will contain the depth contours of the point set P as the boundaries between colored regions.

The algorithm generates a discrete image of the depth contours and thus aliasing is possible. The three main sources of error are: (1) sub-pixel precision in the input (i.e. the coordinates of the input points are not integers), (2) computing the depth of a pixel in the dual plane, and (3) rendering the lines back in the primal. Krishnan *et al* prove that the depth contours produced err by at most one pixel. Such a small error is acceptable for most applications and the runtime speedup is usually worth the small discrepancy.

The complexity of this algorithm is $O(nm + nm^{1/2} + m^{3/2}) = O(nm + m^{3/2})$, where m is the number of pixels in the output image. The first term corresponds to the first stage of the algorithm – n half-planes are rendered, each of which covers $O(m)$ pixels. The second term corresponds to the second stage of the algorithm, where n lines are rendered, and each of them has a length of $O(m^{1/2})$. The final term also corresponds to the second stage of the algorithm – there are a maximum of m pixels that represent primal lines that are rendered, and each of them will cover $O(m^{1/2})$ pixels when this happens.

3. Our Algorithm

Our algorithm is an improvement of the image-space algorithm of Krishnan *et al* [KMV2002]. That algorithm renders a 3D line in the primal image for each pixel drawn in the dual plane, and these lines cover regions in the primal image plane. However, the following simple observation results in a number of improvements: The union of primal lines corresponding to the points along a *dual line segment* between two adjacent vertices of the arrangement form a double wedge, which may be rendered as two triangles at a fixed depth. The accurate identification of these dual line segments is the only difficult part.

The improvement is threefold: First, the number of rendering operations is dramatically decreased. We render a reasonable number of triangles ($O(n^2)$) instead of a huge number of lines ($O(m)$). This improvement is particularly significant if n is much smaller than m. Second, the accuracy of the result is improved. Since we render an entire triangular region instead of covering it by lines, we

are less prone to aliasing problems if the lines are not dense enough. Third, we only have to deal with $O(nm^{1/2})$ pixels from the dual plane renderings, rather than the full m pixels required by the original algorithm. This also implies that to make the best use of this algorithm, the number of sites should be no greater than $m^{1/2}$. This is reasonable, as having more sites than that results in a high density of depth contours relative to the number of pixels, meaning that the resolution of the image is too small to support the required amount of detail.

3.1 Detecting dual line segments

The main challenge in implementing this algorithm is accurately detecting the dual line segments. Because the lines have been rasterized, it can be difficult to determine exactly which direction a given line is heading, or even whether there is a line segment between two neighboring pixels. We deal with the first issue in a fairly unsophisticated manner that could be improved upon, and with the second problem by sidestepping it.

We use the following algorithm to trace the line segments in each of the dual planes. First, when rendering just the lines of the bounded duals, use the same stencil buffer operation as used when rendering the half-planes. This will cause the stencil buffer to be 0 wherever no line has been drawn, 1 wherever a line has been drawn with no intersecting lines, and greater than 1 at every point of intersection between two or more lines. Clearly, this allows us to determine where intersections between lines occur, and therefore where a given line segment terminates.

Once we have the stencil buffers from the bounded dual half-planes, the stencil buffers from the bounded dual lines, and the color buffers from the bounded dual lines, we can begin tracing the bounded dual line segments and computing the appropriate set of double wedges. Pseudocode of this procedure appears in Figure 3. First, we find a pixel where a line intersects the edge of the bounded dual. We take that pixel as one end of a line segment and add it to an event queue, and then we walk along the line by finding a neighboring pixel that is "colored in" (i.e., has had a line rasterized there). As we walk along the line, we erase it so that we do not consider it again in the future, thereby guaranteeing termination of the algorithm. We continue until we encounter a pixel whose stencil buffer value is greater than 1, at which point we flag that pixel as the end of the line segment, and create the appropriate double wedge, assigning the depth found in the half-plane stencil buffer along that segment between the endpoints (the pixels at the endpoints may have a different value for the level, so we do not use them). Every time an endpoint is found, it is added to the event queue, and every time a double wedge is created, we check if it has any neighboring pixels that are colored (thereby signifying that there is some line segment leading from that endpoint that has not been considered). If so, we start a new double wedge from that end point. If not, we go to the next event in the queue. We repeat this until the queue is empty, at which point we look for the next pixel where a line segment intersects the edge of the bounded dual. We repeat this until there are no more line segments intersecting the edge of the bounded dual. Once we have completed this for both bounded duals, we draw the set of double wedges at the appropriate depths.

3.2 Caveats

Problems with our algorithm can arise near points of intersection – the rasterizations of two lines will intersect at some set of pixels, but there may be places near those intersection pixels where the rasterizations border each other, making it difficult to determine which direction the current line segment is heading from a given pixel. The wrong choice can easily result in an incorrect double wedge, as the point of intersection between the two lines in question can be completely missed, resulting in a "line segment" that has a corner. We deal with this problem by limiting ourselves to only two or three of the eight neighboring pixels when walking along a line. Specifically, the first colored pixel we find adjacent to an endpoint reduces the set of directions we can travel from eight to three, then the next pixel we see in that direction that is either immediately clockwise or counter-clockwise from the first direction further limits the set of directions to those two directions. For example, if we are at an intersection point, and we find a colored pixel west of the intersection, we will continue west, but always checking northwest, west, and southwest for colored pixels. At some point before finding an intersection, we may encounter a colored pixel to the northwest, at which point we will go northwest, and now only consider pixels west and northwest of the current pixel. All other directions will be ignored. These tests are found in the GetNextIntersection function in the pseudocode in Figure 3. This works in most cases, but it is still possible to design arrangements of rasterized lines for which it will fail – specifically, if the intersecting lines form an obtuse angle with each other, the algorithm can follow the first line, then start following the second line which has the same pair of rules (e.g., always go west or northwest), and thereby miss the actual intersection point. It is not difficult to conceive an algorithm that would follow the rasterized lines more accurately (for instance, maintaining a vector from the first endpoint to the current point in the line in order to make a guess as to which neighboring pixel is more likely to have the correct point), but most or all of these algorithms will suffer from the same weakness to some degree, simply because the problem is not well-conditioned; we cannot always know which direction the rasterized line is going without rasterizing it ourselves, which defeats the purpose of using the graphics hardware.

Similarly, if we are at a point of intersection, and a neighboring pixel is also a point of intersection, we cannot know if there is actually a line segment connecting the two points, or if they are simply neighboring points of intersection with no direct line segment between them. Fortunately, in this case we can simply fall back on the original algorithm, and draw lines in the primal plane for each of those intersection points, instead of double wedges. This can be seen in the function WalkArrangement in the pseudocode in Figure 3.

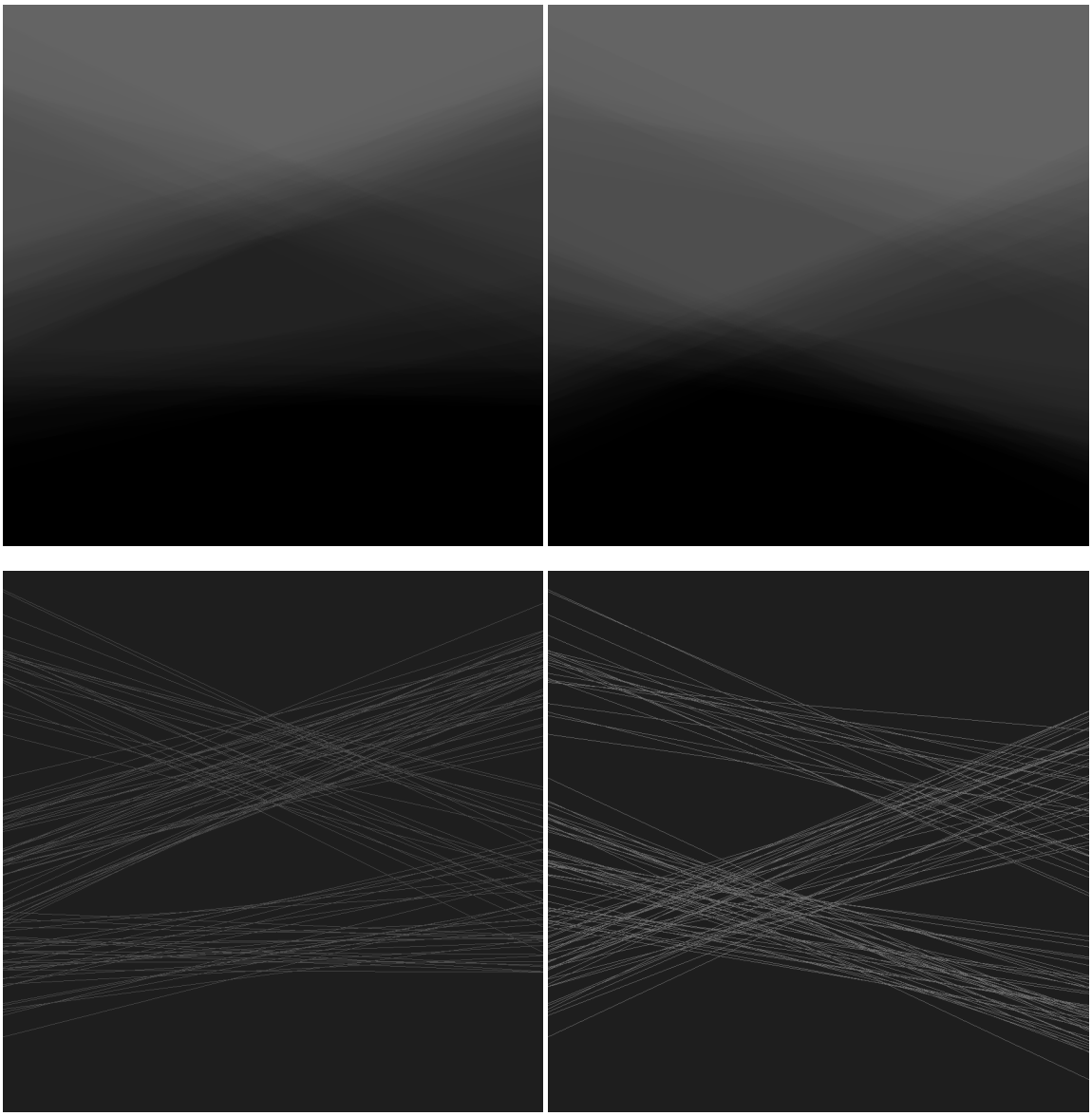


Figure 2: *The two bounded duals (left and right) of the set of sites of Figure 1. (top) Set of half-planes with lighter areas having a higher level (i.e., more half-planes drawn there). (bottom) corresponding set of lines. Note that the clustering of lines in the dual arrangement corresponds to the clustering of points in the primal plane.*

3.3 Complexity

Our line-tracing algorithm runs in $O(nm^{1/2})$ time – for every dual line, we are going to have to walk through $O(m^{1/2})$ pixels. There will be $O(n^2)$ double wedges that we have to render – an arrangement of n lines has $O(n^2)$ line segments – so drawing the double wedges requires $O(n^2)$ time on the CPU. It can also be seen quite easily (by induction) that for each site, drawing the double wedges associated with that site results in exactly (up to rasterization error) m pixels being covered between all of the wedges. Thus, rasterizing the $O(n^2)$ double wedges takes $O(nm)$ time. The maximum number of primal lines we can draw due to the problem of neighboring intersection points is $O(n^2)$, since those can only be drawn from intersection points, which gives a total complexity of $O(n^2m^{1/2})$. The initial steps of the algorithm (drawing the dual half-planes and dual lines) has the same complexity in our algorithm as in the original – $O(nm + nm^{1/2})$. Thus, the total worst-case complexity of our algorithm is $O(nm + n^2m^{1/2})$. In the worst case that n is $O(m^{1/2})$, this simplifies to the same complexity as the algorithm of Krishnan *et al*. In the more realistic case that $n = o(m^{1/2})$, our algorithm is asymptotically faster. This claim is backed by experimental results, shown below.

```
Function ComputeDoubleWedges
   while there is a pixel p of a line intersecting the edge of
         the BD
      WalkArrangement(p)
   endwhile
```

```
Function WalkArrangement(Pixel p)
   Queue events
   Push(events, p)
   while there is an event p to process
      Pixel newp
      depth = GetNextIntersection(p, newp)
      if p != newp
         if p is adjacent to newp
            // we don't create a double wedge because there
            // may not have been a line between two
            // adjacent intersection points
            AddSegment(newp, depth)
         else
            AddDoubleWedge(p, newp, depth)
         endif
         Push(events, newp)
      endif
      if there are no more lines adjacent to p
         Pop(events, p)
      endif
   endwhile
```

```
Function GetNextIntersection(Pixel p, Pixel nextp)
   nextp = p
   direction = NONE
   List depthList
   while nextp is within the image and
         (StencilBuffer(nextp) < 2 or nextp == p)
      erase color at nextp
      for each of the eight directions, if direction contains
            that direction or direction equals NONE or
            direction does not contain any of the non-
            adjacent directions
         if the pixel in that direction is part of a line
            direction |= that direction
            Add(depthList, Depth(nextp))
            nextp = nextp + offset in that direction
         endif
      endfor
   endwhile
   return most frequent depth in depthList
```

```
Function AddSegment(Pixel p, depth d)
   // add the primal line segment that corresponds to dual
   // point p to the list of line segments to draw at depth d
```

```
Function AddDoubleWedge(Pixel first, Pixel last,
      depth d)
   // Add the double wedge formed by the two primal
   // lines that correspond to the two dual points to the list
   // of double wedges to draw at depth d
```

Figure 3: *Pseudo-code of our wedge-drawing algorithm.*

4. Experimental Results

To compare our image-space algorithm with that of Krishnan *et al* [KMV2002], we implemented and ran both on a variety of test cases. Both may be implemented quite easily in C++ and OpenGL. However, a couple of important implementation details are missing from the paper of Krishnan *et al*, worth mentioning here. First, and most important, was the transformation of a point in BD2 (the second bounded dual) back to a line in the primal plane. Without this the algorithm is meaningless, as the standard type of transformation would give incorrect lines in the primal plane. Thus a point p in BD2 reverts to the line $y = x/p_x - p_y/p_x$. In the implementation of Krishnan *et al*, they simply rotate BD2 by 90 degrees to get the same effect.

A second issue is one of coverage of the primal plane with the lines corresponding to points from the two bounded duals. Krishnan *et al* do not discusses why, or even whether, the algorithm will result in full coverage of the primal plane, or if the lines of one depth contour will completely occlude the lines of contours deeper than it, and be completely occluded by the lines of contours shallower than it. In practice, the primal plane is completely covered after only two or three sites. However, we are not so lucky when trying to cover deeper levels, so often there are significant artifacts where a given level shows the color of the next deeper level because the level was not fully covered. See Figure 1 for an example of inadequate coverage. Ultimately, the only way to deal with this problem was to draw the lines at higher density, i.e. draw multiple lines per pixel by "supersampling" the pixel. If a pixel is colored in the dual arrangement, we subdivide it into a regular grid of s by s subpixels, each of which we consider to also be colored, thus generating s^2 primal lines at this depth. Since we do not actually know which line contributed to this pixel, we cannot do true supersampling, where only a subset of these pixels would contribute to the line. For resolutions up to 1400 x 1000, a supersampling rate of 4x4 gives a fairly clean image. However, the supersampling rate is dependent on both m and n, so no single value of s will give a good balance between speed and quality. Krishnan *et al* deal with the coverage problem in their implementation by increasing the line thickness, but this seems also to be a significant hidden source of inaccuracy – this causes lines to be wider than one pixel, which is equivalent to saying that a point in the dual plane transforms to an oriented rectangle of infinite length and finite width in the primal plane, rather than a line.

An alternative solution that would improve accuracy over both approaches outlined above would be to simply increase the resolution of the dual plane renderings. However, this adds quite a bit of complexity to the algorithm, especially since limitations in the resolution of the buffers on the graphics cards would prevent even a doubling of the vertical resolution while still expecting a reasonable output resolution. For example, to do true supersampling of the dual planes by a factor of two in each dimension for an output image of size 1024 x 1024 would require a frame buffer of size 2048 x 2048, but on many systems, the maximum vertical resolution will be less than 2048. Thus, to achieve that resolution, all renderings of each of the dual planes would have to be done twice – once each for the top and bottom halves of the dual planes. Obviously, this also adds a fairly significant constant to the

rendering times of the algorithm, making it less than ideal for an approximation algorithm.

We ran the two algorithms on an Intel Centrino 1.8GHz processor with 1GB of RAM and a 128MB RAM ATI FireGL128 video card. All test were completed at a resolution 1000 by 1000 pixels. Due to the nature of this algorithm, it is not reasonable to amortize the cost of sending the geometry over the graphics bus across a series of frames, so our test results are the average of single frames from 10 runs on different randomly generated sites at 100, 250, 500, 750, 1000, and 2500 sites. In order to maintain objectivity of the data collected, each set of sites was used for both algorithms.

As can be seen in Figure 4, our algorithm outperforms the algorithm of [KMV2002] by a statistically significant amount until the number of sites rises above the square root of the number of pixels, at which point they become essentially equivalent. This is due to the fact that almost all of the intersection pixels examined by the arrangement-walking algorithm have another intersection adjacent to them – the density of intersections in the dual-planes is too high – so our algorithm simply draws lines across most of the primal plane rather than double-wedges.

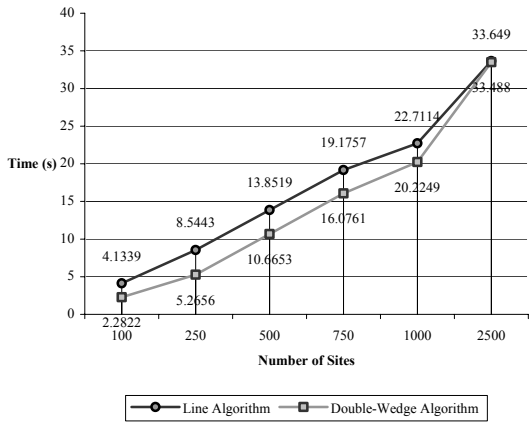


Figure 4: *Average results over 10 runs each for the two different image space algorithms. The screen resolution was 1000 by 1000 pixels. The sites were randomly generated per pair of tests. As expected by the complexity analysis, above* $n = m^{1/2}$ *the two algorithms converge.*

5. Conclusion

Our fairly simple improvements to the algorithm of Krishnan *et al* allow us to more quickly visualize the depth contours and Tukey median of a set of sites in 2D. They also allow us to visualize the depth contours in more detail, as the algorithm shows an increasing relative efficiency as the resolution of the rendered image increases while the number of sites remains constant. Indeed, our algorithm is now fast enough that we can view interactive changes to the depth contours for a small set of dynamic sites.

There are a number of interesting possible future directions for this problem. First, given what we have shown, is it possible to more efficiently compute only the Tukey median? The Tukey median only depends directly on the deepest depth contour, so if it is possible to draw only that contour while avoiding much of the computation currently required to approximate it, then we could visualize the Tukey median much more quickly. Second, it would be interesting to consider whether these techniques can be extended to 3D using arrangements of planes rather than arrangements of lines and then drawing slices of the volume to visualize the depth contours and approximate the Tukey median. Finally, are there further possible refinements that would allow us to interactively visualize larger sets of sites? With current high-end hardware, it may be possible to render the depth contours interactively for about 100 sites using our algorithm, but there may be further refinements that could dramatically increase that number without waiting for hardware speeds to increase dramatically.

References

[CSY1987] R. COLE, M. SHARIR, AND C. K. YAP. "On *k*-hulls and related problems". *SIAM Journal on Computing 15*, 1 (1987), 61–77.

[KMV2002] S. KRISHNAN, N. MUSTAFA, AND S. VENKATASUBRAMANIAN, "Hardware-assisted computation of depth contours". *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms* (2002).

[MRR*2001] K. MILLER, S. RAMASWAMI, P. ROUSSEEUW, T. SELLARES, D. SOUVAINE, I. STREINU AND A. STRUYF. "Fast implementation of depth contours using topological sweep". *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms* (2001), pp. 690–699.

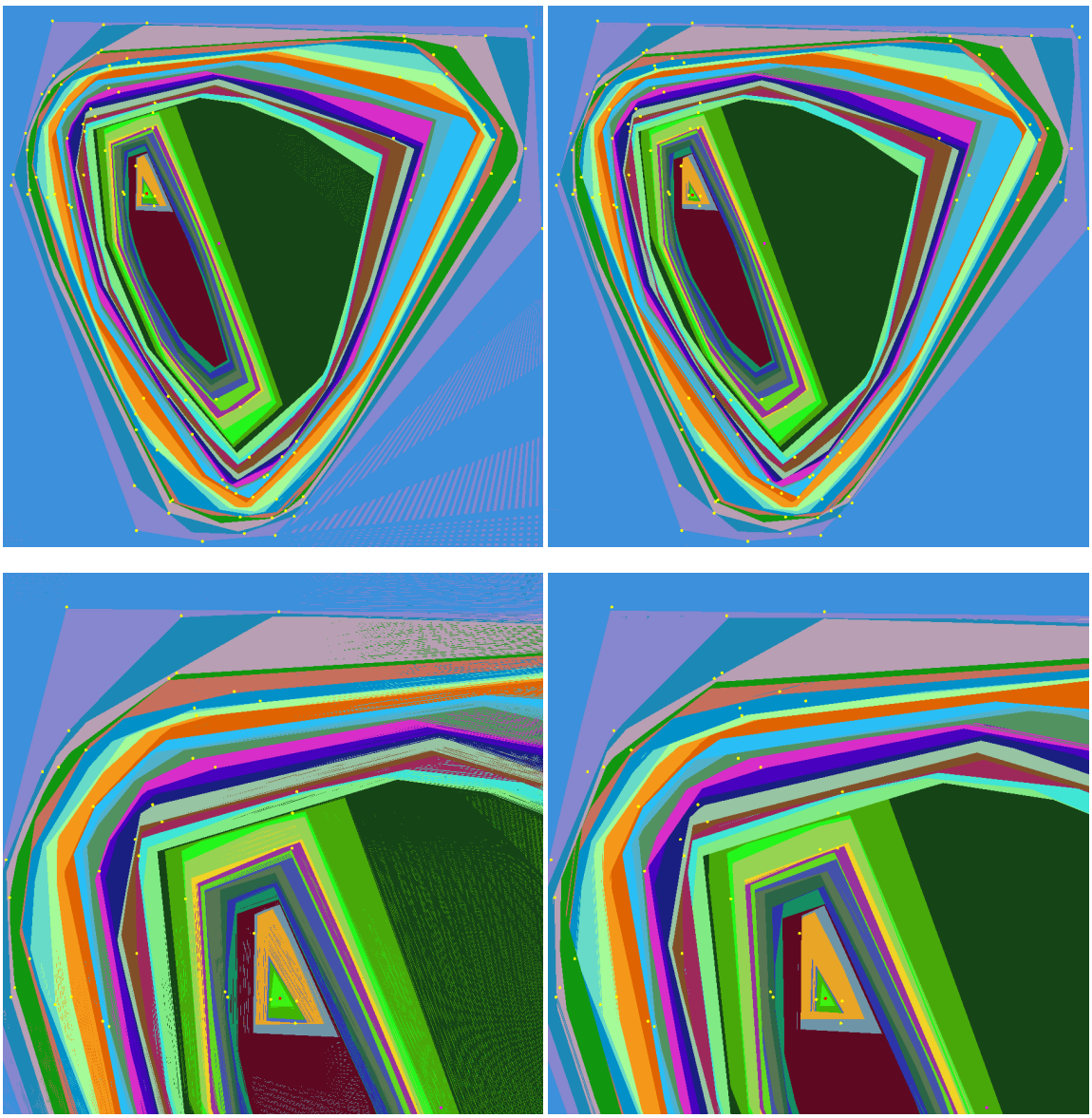


Figure 1: *Comparison of depth contours generated by our implementations of the image-space algorithm of [KMV2002] (left) with our image-space algorithm (right). The top images are of the entire region of interest in the primal plane, containing 100 sites. The lower images are zooms on the cluster of sites in the upper-left corner of the primal plane. In all four images, the yellow points are the sites, the magenta point is the mean of the sites, and the red point near the upper-left cluster is the Tukey median of the sites. The algorithm of [KMV2002] generated 637,310 primal lines using a supersampling rate of 2, compared to 64,324 lines and 15,398 double wedges at the same supersampling rate for our algorithm. Our algorithm runs about twice as fast for this data set at a resolution of 900 by 900 pixels. Note that both algorithms have some issues achieving a full coverage of the image plane at a given level, which allows deeper depth contours to show through at some pixels. This is the main source of noise in all four images. Note, however, that the noise is much worse in [KMV2002] because the regions are covered by lines, whereas in our algorithm the regions are covered by polygons. At a supersampling rate of 4, the noise disappears for [KMV2002], but that generates 2,549,596 lines and runs about three times slower than our algorithm at the lower supersampling rate.*